Excommunication: Transforming π-Calculus Specifications to Remove Internal Communication

G.W. Hamilton*, B. Aziz†

*School of Computing, Dublin City University, Dublin, Ireland
†School of Computing, University of Portsmouth, Portsmouth, United Kingdom

Abstract. In this paper, we present a new automatic transformation algorithm called *excommunication* that transforms π-calculus processes to remove parallelism, and hence internal communication. We prove that the transformation is correct and that it always terminates for any specification in which the named processes are in a particular syntactic form we call *serial form*. We argue that this transformation facilitates the proving of properties of mobile processes, and demonstrate this by showing how it can be used to simplify a *leakage analysis*.

1 Introduction

Unfold/fold program transformation techniques were first defined for functional languages by Burstall and Darlington [4] and have since been studied extensively for both functional and logic programs [17, 18, 20, 16, 8, 14, 10]. However, these techniques have rarely been applied to concurrent languages (some notable exceptions are [19, 15, 5, 6] and the partial evaluation approaches in [11, 7, 3]). This is partly due to the fact that the non-determinism and synchronisation mechanisms used in concurrent languages substantially complicate their semantics, and thus their corresponding transformation. However, such transformation can be very useful for concurrent languages, since the resulting programs will be easier to analyse and prove properties about as we do not have to deal with the complications of parallel composition and internal communication.

In this paper, we present an automatic transformation algorithm called *excommunication* that transforms π-calculus [12, 13] specifications to remove parallelism, and hence internal communication.

Example 1. As a simple example of applying the excommunication algorithm, consider the following π-calculus specification:

$$(\nu m)(B\lfloor l, m\rfloor | B\lfloor m, r\rfloor)$$
where
$$B \triangleq (i, o).i(x).\bar{o}x.B\lfloor i, o\rfloor$$

This is the definition of two single cell buffers that are chained together. One cell receives its input at l, and emits this at m. The other cell receives its input at m,

and emits this at r. This specification is transformed by the excommunication algorithm to the following:

$$
\begin{array}{l}
C\lfloor l, r\rfloor \\
\textbf{where} \\
C \triangleq (l, r).l(x).D\lfloor l, r, x\rfloor \\
D \triangleq (l, r, x).\overline{r}x.C\lfloor l, r\rfloor + [l = r]D\lfloor l, r, x\rfloor + l(y).\overline{r}x.D\lfloor l, r, y\rfloor
\end{array}
$$

The named process C corresponds to a state in which both single cell buffers are empty, and the named process D corresponds to a state in which one of the buffers is full. All parallelism, and hence all internal communication, has been removed by the transformation. The resulting specification is equivalent to the original one, which means that it will have the same observable behaviour within all contexts. This is why the definition of the named process D contains a match between l and r; if the process appears in a context in which the same name is substituted for l and r, then the contents of the second buffer can be passed directly to the first buffer.

The excommunication algorithm for the π-calculus that we present here is inspired by and similar in form to the deforestation algorithm for the λ-calculus [20, 8], except that the goal here is to remove parallelism and hence internal communication rather than the intermediate structures that are removed by deforestation. We prove that the transformation is correct and that it always terminates for any specification in which the named processes are in a particular syntactic form we call *serial form*. We argue that this transformation facilitates the proving of properties of mobile processes as we do not have to deal with the complications of parallel composition and internal communication, and demonstrate this by showing how it can be used to simplify a *leakage analysis*.

The remainder of this paper is structured as follows: In Section 2, we define the syntax and semantics of the π-calculus. In Section 3, we define serial form and the excommunication algorithm, and prove the excommunication theorem which states that excommunication is correct and always terminates for any specification in which the named processes are in serial form. In Section 4, we show how excommunication facilitates the static analysis of π-calculus specifications by defining a leakage analysis on the simplified processes resulting from transformation. Section 5 concludes and considers possible further work.

2 The π-Calculus

In this section, we describe the syntax and semantics of the π-calculus.

Definition 1 (Syntax of the π-Calculus). The syntax of the π-calculus is shown in Fig. 1. □

A specification consists of a process and a number of named process definitions. A process can be null, a prefix action (input, output or silent), match, restriction, non-deterministic choice, parallel composition or named process application. A match $[x = y]P$ only proceeds as process P if the names substituted

$$
\begin{array}{lll}
S & ::= P \textbf{ where } D_1 \ldots D_n & \text{Specification} \\
\\
D & ::= p \stackrel{\triangle}{=} (x_1 \ldots x_n).P & \text{Named Process Definition} \\
\\
P, Q & ::= \mathbf{0} & \text{Null Process} \\
 & \mid x(y).P & \text{Input} \\
 & \mid \overline{x}y.P & \text{Output} \\
 & \mid \tau.P & \text{Silent Action} \\
 & \mid [x = y]P & \text{Match} \\
 & \mid (\nu x)P & \text{Restriction} \\
 & \mid P + Q & \text{Non-Deterministic Choice} \\
 & \mid P \mid Q & \text{Parallel Composition} \\
 & \mid p\lfloor x_1 \ldots x_n \rfloor & \text{Named Process Application}
\end{array}
$$

Fig. 1. Syntax of the π-Calculus

for x and y are the same; otherwise the process blocks. A named process definition has a number of parameters $x_1 \ldots x_n$ and a process body defined over these parameters. Named process parameters, input variables and restricted variables are *bound* within a process. A name is *free* in a process if it is not bound. We denote the set of names which are free in process P by $fn(P)$. We write $P \equiv Q$ if P and Q differ only in the names of bound variables and are therefore α-equivalent.

Definition 2 (Renaming). We use the notation $\sigma = \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ to denote a *renaming*. If P is a process, then $P\sigma = P\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ is the result of simultaneously replacing the free names $x_1 \ldots x_n$ with the corresponding names $x'_1 \ldots x'_n$ respectively, in the process P while ensuring that bound names are renamed appropriately to avoid name capture. □

We define (possibly non-terminating) reduction rules for the π-calculus that reduce processes to the following *normal form* that contains no parallel composition or named processes.

Definition 3 (Normal Form). Normal form is defined as shown in Fig. 2.

$$
\begin{array}{lll}
P, Q & ::= \mathbf{0} \\
 & \mid x(y).P \\
 & \mid \overline{x}y.P \\
 & \mid \tau.P \\
 & \mid [x = y]P \\
 & \mid (\nu x)P \\
 & \mid P + Q
\end{array}
$$

Fig. 2. Normal Form

Definition 4 (Reduction Rules). The reduction rules for the π-calculus are given in Fig. 3.

(1) $\mathcal{R}[\![P \textbf{ where } D_1 \dots D_n]\!] = \mathcal{R}_{\mathcal{P}}[\![P]\!][\![\mathbf{0}]\!]\ \{\}\ \{D_1 \dots D_n\}$

(2) $\mathcal{R}_{\mathcal{P}}[\![[x=y]P]\!][\![Q]\!]\ \theta\ \Delta = \mathcal{R}_{\mathcal{P}}[\![Q]\!][\![[x=y]P]\!]\ \theta\ \Delta$
$= \begin{cases} \mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta, & \text{if } x = y \\ \mathbf{0}, & \text{if } x \in \theta \vee y \in \theta \\ [x=y](\mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta), & \text{otherwise} \end{cases}$

(3) $\mathcal{R}_{\mathcal{P}}[\![(\nu x)P]\!][\![Q]\!]\ \theta\ \Delta = \mathcal{R}_{\mathcal{P}}[\![Q]\!][\![(\nu x)P]\!]\ \theta\ \Delta$
$= \mathcal{R}_{\mathcal{P}}[\![P[x'/x]]\!][\![Q]\!]\ (\theta \cup \{x'\})\ \Delta\ (x' \notin fn(Q))$

(4) $\mathcal{R}_{\mathcal{P}}[\![P_1 + P_2]\!][\![Q]\!]\ \theta\ \Delta = \mathcal{R}_{\mathcal{P}}[\![Q]\!][\![P_1 + P_2]\!]\ \theta\ \Delta$
$= (\mathcal{R}_{\mathcal{P}}[\![P_1]\!][\![Q]\!]\ \theta\ \Delta) + (\mathcal{R}_{\mathcal{P}}[\![P_2]\!][\![Q]\!]\ \theta\ \Delta)$

(5) $\mathcal{R}_{\mathcal{P}}[\![P_1|P_2]\!][\![Q]\!]\ \theta\ \Delta = \mathcal{R}_{\mathcal{P}}[\![Q]\!][\![P_1|P_2]\!]\ \theta\ \Delta$
$= \mathcal{R}_{\mathcal{P}}[\![\mathcal{R}_{\mathcal{P}}[\![P_1]\!][\![P_2]\!]\ \{\}\ \Delta]\!][\![Q]\!]\ \theta\ \Delta]\!]$

(6) $\mathcal{R}_{\mathcal{P}}[\![p\lfloor x_1 \dots x_n \rfloor]\!][\![Q]\!]\ \theta\ \Delta = \mathcal{R}_{\mathcal{P}}[\![Q]\!][\![p\lfloor x_1 \dots x_n \rfloor]\!]\ \theta\ \Delta$
$= \mathcal{R}_{\mathcal{P}}[\![unfold(p\lfloor x_1 \dots x_n \rfloor, \Delta)]\!][\![Q]\!]\ \theta\ \Delta$

(7) $\mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta = (\mathcal{R}_{\mathcal{L}}[\![P]\!][\![Q]\!]\ \theta\ \Delta) + (\mathcal{R}_{\mathcal{L}}[\![Q]\!][\![P]\!]\ \theta\ \Delta)$

(8) $\mathcal{R}_{\mathcal{L}}[\![\overline{x}y.Q]\!][\![x'(z).P]\!]\ \theta\ \Delta$
$= \begin{cases} \mathbf{0}, & \text{if } x \in \theta \\ (\nu y)((\mathcal{R}_{\mathcal{P}}[\![[x=x']P]\!][\![Q[y/z]\!]\ \theta\ \Delta) + \overline{x}y.(\mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta)), & \text{if } y \in \theta \\ (\mathcal{R}_{\mathcal{P}}[\![[x=x']P]\!][\![Q[y/z]\!]\ \theta\ \Delta) + \overline{x}y.(\mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta), & \text{otherwise} \end{cases}$

(9) $\mathcal{R}_{\mathcal{L}}[\![\overline{x}y.P]\!][\![Q]\!]\ \theta\ \Delta$
$= \begin{cases} \mathbf{0}, & \text{if } x \in \theta \\ (\nu y)\overline{x}y.(\mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta), & \text{if } y \in \theta \\ \overline{x}y.(\mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta), & \text{otherwise} \end{cases}$

(10) $\mathcal{R}_{\mathcal{L}}[\![x(y).P]\!][\![Q]\!]\ \theta\ \Delta$
$= \begin{cases} \mathbf{0}, & \text{if } x \in \theta \\ x(y').(\mathcal{R}_{\mathcal{P}}[\![P[y'/y]]\!][\![Q]\!]\ \theta\ \Delta), & \text{otherwise } (y' \notin fn(x(y).P|Q)) \end{cases}$

(11) $\mathcal{R}_{\mathcal{L}}[\![\tau.P]\!][\![Q]\!]\ \theta\ \Delta = \tau.(\mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta)$

(12) $\mathcal{R}_{\mathcal{L}}[\![\mathbf{0}]\!][\![Q]\!]\ \theta\ \Delta = \mathbf{0}$

Fig. 3. Reduction Rules for π-Calculus

These rules closely mirror the denotational semantics for the π-calculus given in [1]. Rule (1) of the form $\mathcal{R}[\![S]\!]$ defines the reduction of the specification S.

Rules (2)-(7) of the form $\mathcal{R}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \theta\ \Delta$ define the reduction rules for the parallel composition of the processes P and Q. The parameter θ contains the set of names that are restricted to one of the two processes, and can therefore not be used as a communication channel between them unless there is a scope extrusion. The parameter Δ contains the set of named process definitions. The reduction rules are followed in a top-down order, so rules (2)-(6) are firstly followed to reduce matching, restriction, choice, parallel composition and named process application.

In rule (2), if matching can be performed then it is removed. If either of the names being matched is restricted then the matching fails, otherwise the match remains. In rule (3), a restriction is removed and the restricted name is added to θ; this is renamed so as not to clash with the free names of the other process. In rule (4), a choice is distributed across the parallel composition. In rule (5), a parallel composition in one process is reduced before further reducing the surrounding composition. In rule (6), a named process application is unfolded. The function *unfold* replaces a named process application with the process body, with the formal names of the body replaced by the actual names in the application. This is defined more formally as follows:

$$unfold(p\lfloor x_1 \dots x_n \rfloor, \Delta) = P[x_1/x'_1, \dots, x_n/x'_n] \text{ (where } (x'_1 \dots x'_n).P \in \Delta)$$

When none of the rules (2)-(6) apply, both processes will be either $\mathbf{0}$ or prefixed by an action. Rule (7) is then applied to give a choice of two possible left-prioritised parallel compositions for each possible ordering of the two processes. In a left-prioritised parallel composition, the left process must first perform an action before the residue is composed with the right process. Rules (8)-(12) of the form $\mathcal{R}_{\mathcal{L}}[\![P]\!][\![Q]\!]\ \theta\ \Delta$ define the left-prioritised parallel composition of processes P and Q, where the next action must be performed by process P.

In rule (8), if the left process is prefixed by an output action and the right by an input action, then communication may or may not take place. If the output channel is restricted to the left process, then the communication cannot take place so the result is $\mathbf{0}$. If communication does take place, then the names of the channels in the two actions must be the same so a matching operation is created and the residues of the two processes are composed. Otherwise, the output action is retained and the residue composed with the right process. If the output name is restricted to the left process its scope is extruded, so a restriction is added.

In rule (9), if the left process is prefixed by an output action but the right process is not prefixed by an input action, then if the output channel is restricted to the left process, communication cannot take place so the result is $\mathbf{0}$. Otherwise, the output is retained and the residue composed with the right process. If the output name is restricted to the left process, then this is a bounded output, so a restriction is added.

In rule (10), if the left process is prefixed by an input action, then if the input channel is restricted to the left process, communication cannot take place so the result is $\mathbf{0}$. Otherwise, the input is retained and the residue composed with the right process where the input name is renamed so as not to clash with the free names of the right process. In rule (11), if the left process is prefixed

by a silent action, then the action is retained and the residue composed with the right process. In rule (12), if the left process is null, then no action can be performed so the result is also null.

3 The Excommunication Algorithm

In this section, we present the excommunication algorithm. This is a set of transformation rules (similar in form to the deforestation algorithm for the λ-calculus [20, 8]) that convert a given process into an equivalent process from which parallel composition, and hence internal communication, has been removed.

The input to the algorithm is a specification in which all named processes are in *serial form*. Processes in serial form contain no parallel composition and therefore no internal communication.

Definition 5 (Serial Form). Serial form is defined as shown in Fig. 4.

$$
\begin{aligned}
P, Q ::= \; & \mathbf{0} \\
& \mid x(y).P \\
& \mid \overline{x}y.P \\
& \mid \tau.P \\
& \mid [x = y]P \\
& \mid (\nu x)P \\
& \mid P + Q \\
& \mid p\lfloor x_1 \ldots x_n \rfloor
\end{aligned}
$$

Fig. 4. Serial Form

Note that the top-level process in the input specification may still contain parallel compositions and internal communication; it is these that are removed by the excommunication algorithm.

3.1 Transformation Rules

The transformation rules for excommunication are very similar to the reduction rules defined in Fig. 3.

Definition 6 (Excommunication Algorithm). The transformation rules for the excommunication algorithm are shown in Fig. 5. □

Rule (1) of the form $\mathcal{T}[\![S]\!]$ defines the transformation of the specification S. Rules (2)-(7) of the form $\mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta$ define the transformation rules for the parallel composition of the processes P and Q. The parameter ρ contains a set of previously encountered memoised processes. The parameter θ contains the set of names that are restricted to one of the two processes, and can therefore

(1) $\mathcal{T}[\![P \textbf{ where } D_1 \dots D_n]\!] = \mathcal{T}_{\mathcal{P}}[\![P]\!][\![\mathbf{0}]\!]\ \{\}\ \{\}\ \{D_1 \dots D_n\}$

(2) $\mathcal{T}_{\mathcal{P}}[\![[x=y]P]\!][\![Q]\!]\ \rho\ \theta\ \Delta = \mathcal{T}_{\mathcal{P}}[\![Q]\!][\![[x=y]P]\!]\ \rho\ \theta\ \Delta$
$= \begin{cases} \mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta, & \text{if } x = y \\ \mathbf{0}, & \text{if } x \in \theta \vee y \in \theta \\ [x=y](\mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta), & \text{otherwise} \end{cases}$

(3) $\mathcal{T}_{\mathcal{P}}[\![(\nu x)P]\!][\![Q]\!]\ \rho\ \theta\ \Delta = \mathcal{T}_{\mathcal{P}}[\![Q]\!][\![(\nu x)P]\!]\ \rho\ \theta\ \Delta$
$= \mathcal{P}[\![P[x'/x]]\!][\![Q]\!]\ (\theta \cup \{x'\})\ \Delta\ (x' \notin fn(Q))$

(4) $\mathcal{T}_{\mathcal{P}}[\![P_1 + P_2]\!][\![Q]\!]\ \rho\ \theta\ \Delta = \mathcal{T}_{\mathcal{P}}[\![Q]\!][\![P_1 + P_2]\!]\ \rho\ \theta\ \Delta$
$= (\mathcal{T}_{\mathcal{P}}[\![P_1]\!][\![Q]\!]\ \rho\ \theta\ \Delta) + (\mathcal{T}_{\mathcal{P}}[\![P_2]\!][\![Q]\!]\ \rho\ \theta\ \Delta)$

(5) $\mathcal{T}_{\mathcal{P}}[\![P_1|P_2]\!][\![Q]\!]\ \rho\ \theta\ \Delta = \mathcal{T}_{\mathcal{P}}[\![Q]\!][\![P_1|P_2]\!]\ \rho\ \theta\ \Delta$
$= \mathcal{T}_{\mathcal{P}}[\![\mathcal{T}_{\mathcal{P}}[\![P_1]\!][\![P_2]\!]\ \{\}\ \{\}\ \Delta]\!][\![Q]\!]\ \rho\ \theta\ \Delta]\!]$

(6) $\mathcal{T}_{\mathcal{P}}[\![p\lfloor x_1 \dots x_n \rfloor]\!][\![Q]\!]\ \rho\ \theta\ \Delta = \mathcal{T}_{\mathcal{P}}[\![Q]\!][\![p\lfloor x_1 \dots x_n \rfloor]\!]\ \rho\ \theta\ \Delta$
$= \begin{cases} P\sigma, \text{ if } \exists (P = Q') \in \rho, \sigma. Q'\sigma \equiv (p\lfloor x_1 \dots x_n \rfloor | Q) \\ p'\lfloor x'_1 \dots x'_k \rfloor \textbf{ where } p' \stackrel{\triangle}{=} (x'_1 \dots x'_k).P', \text{ otherwise} \\ \text{where} \\ \{x'_1 \dots x'_k\} = fn(p\lfloor x_1 \dots x_n \rfloor | Q) \setminus \theta \\ P' = \mathcal{T}_{\mathcal{P}}[\![unfold(p\lfloor x_1 \dots x_n \rfloor, \Delta)]\!][\![Q]\!]\ (\rho \cup \{p'\lfloor x'_1 \dots x'_k \rfloor = p\lfloor x_1 \dots x_n \rfloor | Q\})\ \theta\ \Delta \end{cases}$

(7) $\mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta = (\mathcal{T}_{\mathcal{L}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta) + (\mathcal{T}_{\mathcal{L}}[\![Q]\!][\![P]\!]\ \rho\ \theta\ \Delta)$

(8) $\mathcal{T}_{\mathcal{L}}[\![\overline{x}y.Q]\!][\![x'(z).P]\!]\ \rho\ \theta\ \Delta$
$= \begin{cases} \mathbf{0}, & \text{if } x \in \theta \\ (\nu y)((\mathcal{T}_{\mathcal{P}}[\![[x=x']P]\!][\![Q[y/z]\!]\ \rho\ \theta\ \Delta) + \overline{x}y.(\mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta)), & \text{if } y \in \theta \\ (\mathcal{T}_{\mathcal{P}}[\![[x=x']P]\!][\![Q[y/z]\!]\ \rho\ \theta\ \Delta) + \overline{x}y.(\mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta), & \text{otherwise} \end{cases}$

(9) $\mathcal{T}_{\mathcal{L}}[\![\overline{x}y.P]\!][\![Q]\!]\ \rho\ \theta\ \Delta$
$= \begin{cases} \mathbf{0}, & \text{if } x \in \theta \\ (\nu y)\overline{x}y.(\mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta), & \text{if } y \in \theta \\ \overline{x}y.(\mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta), & \text{otherwise} \end{cases}$

(10) $\mathcal{T}_{\mathcal{L}}[\![x(y).P]\!][\![Q]\!]\ \rho\ \theta\ \Delta$
$= \begin{cases} \mathbf{0}, & \text{if } x \in \theta \\ x(y').(\mathcal{T}_{\mathcal{P}}[\![P[y'/y]]\!][\![Q]\!]\ \rho\ \theta\ \Delta), & \text{otherwise } (y' \notin fn(x(y).P|Q)) \end{cases}$

(11) $\mathcal{T}_{\mathcal{L}}[\![\tau.P]\!][\![Q]\!]\ \rho\ \theta\ \Delta = \tau.(\mathcal{T}_{\mathcal{P}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta)$

(12) $\mathcal{T}_{\mathcal{L}}[\![\mathbf{0}]\!][\![Q]\!]\ \rho\ \theta\ \Delta = \mathbf{0}$

Fig. 5. Transformation Rules for Excommunication

not be used as a communication channel between them unless there is a scope extrusion. The parameter Δ contains the set of named process definitions.

Rules (8)-(12) of the form $\mathcal{T}_{\mathcal{L}}[\![P]\!][\![Q]\!]\ \rho\ \theta\ \Delta$ define the left-prioritised parallel composition of processes P and Q, where the next action must be performed by process P.

Any potentially infinite sequence of transformation steps must involve the unfolding of a named process application in rule (6); these processes are therefore memoised by being added to ρ. A new named process is also defined in which the arguments are the names within the process that are not contained in θ (and must therefore be internal to the process). If a renaming of a memoised process in ρ is subsequently encountered, it is replaced by an appropriate application of the previously introduced named process.

Example 2. Consider the transformation of the specification given in Example 1:

$$
\begin{array}{l}
(\nu m)(B\lfloor l, m\rfloor | B\lfloor m, r\rfloor) \\
\textbf{where} \\
B \stackrel{\triangle}{=} (i, o).i(x).\overline{o}x.B\lfloor i, o\rfloor
\end{array}
$$

During the transformation, the process $B\lfloor l, m\rfloor | B\lfloor m, r\rfloor$ is encountered. A new named process C with parameters l and r is defined for this (the name m is internal to this process, so is not included in the parameters). Later in the transformation, the process $B\lfloor l, m\rfloor | B\lfloor m, r\rfloor$ is re-encountered, so a recursive application of the named process C is added.

Similarly, during transformation the process $B\lfloor m, r\rfloor | \overline{m}x.B\lfloor l, m\rfloor$ is encountered. A new named process D with parameters l, r and x is defined for this (m is again internal). Later in the transformation, the processes $B\lfloor m, r\rfloor | \overline{m}x.B\lfloor l, m\rfloor$ and $B\lfloor m, r\rfloor | \overline{m}y.B\lfloor l, m\rfloor$ are encountered, so they are replaced with the corresponding applications of the named process D. The overall result of the transformation is therefore as follows:

$$
\begin{array}{l}
C\lfloor l, r\rfloor \\
\textbf{where} \\
C \stackrel{\triangle}{=} (l, r).l(x).D\lfloor l, r, x\rfloor \\
D \stackrel{\triangle}{=} (l, r, x).\overline{r}x.C\lfloor l, r\rfloor + [l = r]D\lfloor l, r, x\rfloor + l(y).\overline{r}x.D\lfloor l, r, y\rfloor
\end{array}
$$

The excommunication theorem can now be stated as follows.

Theorem 1 (Excommunication Theorem). Every π-calculus specification in which all named process definitions are in serial form can be transformed by the excommunication algorithm into an equivalent process which is in serial form. □

In order to prove that the process generated by excommunication is equivalent to the original process, we need to show the result of reducing the transformed process is the same as that of reducing the original process.

Lemma 1 (On Equivalence). $\mathcal{R}[\![\mathcal{T}[\![P]\!]]\!] \equiv \mathcal{R}[\![P]\!]$ □

Proof

As the process P may actually be non-terminating, the proof is by co-induction. This is fairly straightforward since the rules for excommunication are almost identical to those for reduction, with only the addition of folding.
□

Lemma 2 (On The Form of Output Produced by Excommunication). The output process produced by the excommunication algorithm is in serial form. □

Proof

The proof is by structural induction on the transformation rules. This is fairly straightforward since we can see that there is no parallel composition on the right hand side of the transformation rules.
□

Theorem 2 (Termination of Transformation). The transformation algorithm always terminates.

Proof. In order to prove that the algorithm always terminates, it is sufficient to show that there is a bound on the size of processes which are encountered during transformation. If there is such a bound, then there will be a finite number of processes encountered (modulo renaming of variables), and a renaming of a previous process must eventually be encountered. The algorithm will therefore be guaranteed to terminate.

First of all, it must be defined what is meant by the size of a process:

$$\begin{aligned}
\mathcal{S}[\![\mathbf{0}]\!] &= 1 \\
\mathcal{S}[\![x(y).P]\!] &= 1 + \mathcal{S}[\![P]\!] \\
\mathcal{S}[\![\overline{x}\langle y\rangle.P]\!] &= 1 + \mathcal{S}[\![P]\!] \\
\mathcal{S}[\![(\nu x)P]\!] &= 1 + \mathcal{S}[\![P]\!] \\
\mathcal{S}[\![P + Q]\!] &= \mathcal{S}[\![P]\!] + \mathcal{S}[\![Q]\!] \\
\mathcal{S}[\![p\lfloor x_1 \ldots x_n \rfloor]\!] &= n \\
\mathcal{S}[\![P|Q]\!] &= \mathcal{S}[\![P]\!] + \mathcal{S}[\![Q]\!]
\end{aligned}$$

We then define the level of composition of a process as follows:

$$
\begin{aligned}
\mathcal{C}[\![\mathbf{0}]\!] &= 1 \\
\mathcal{C}[\![x(y).P]\!] &= \mathcal{C}[\![P]\!] \\
\mathcal{C}[\![\overline{x}\langle y\rangle.P]\!] &= \mathcal{C}[\![P]\!] \\
\mathcal{C}[\![(\nu x)P]\!] &= \mathcal{C}[\![P]\!] \\
\mathcal{C}[\![P+Q]\!] &= max(\mathcal{C}[\![P]\!], \mathcal{C}[\![Q]\!]) \\
\mathcal{C}[\![p\lfloor x_1 \ldots x_n \rfloor]\!] &= 1 \\
\mathcal{C}[\![P|Q]\!] &= \mathcal{C}[\![P]\!] + \mathcal{C}[\![Q]\!]
\end{aligned}
$$

We then prove, for an input process with level of composition c and maximum named process size s, that the size of processes encountered during transformation is bounded by $c \times s$. This is done by structural induction on the transformation rules.

4 Example Application: A Leakage Analysis

The excommunication transformation provides us with a simple serial form for the π-calculus on which we can define static analyses to detect any interesting properties of the resulting observable process behaviour. In this section, we give an example of one such analysis that can detect leakages of sensitive data given a security partial ordering relation.

We start first by assuming a partial order of security labels, $(\Xi, \leq_\ell)$, where the set Ξ is ranged over by labels, $\ell, \ell', \ldots$, and is ordered by a partial ordering relation, $\leq_\ell$. We use these labels to annotate all channel names to reflect the trustworthiness of the sub-process owning those channels. Hence, we annotate as $x^\ell(y).P$ and $\overline{x}^{\ell'} z.Q$. In a real scenario, this annotation would be subject to some predefined multi-level security policy derived using a requirements analysis.

We next define an input/output data analysis on an annotated specification as the interpretation function, $\mathcal{A}_S[\![S]\!]\ \phi_I \phi_O \in (\Phi_\perp \times \Phi_\perp)$, where $\Phi_\perp : \mathcal{N} \to \wp(\Xi)$ is the domain of mappings, ϕ, from names to sets of security labels. The bottom element mapping in this domain, $\perp_\Phi$, is defined as the element that maps every name to an empty set of labels:

$$\forall x \in \mathcal{N} : \perp_\Phi(x) = \{\}$$

We also define the union of such mappings, $\uplus_\phi : \Phi_\perp \times \Phi_\perp \to \Phi_\perp$, as follows:

$$\forall x \in \mathcal{N}, \phi_1, \phi_2 \in \Phi_\perp : (\phi_1 \uplus_\phi \phi_2)(x) = \phi_1(x) \cup \phi_2(x)$$

and the pairwise union, $\uplus_{\phi,\phi}$, distributes $\uplus_\phi$ over a pair of mappings:

$$\forall \phi_1, \phi_1', \phi_2, \phi_2' \in \Phi_\perp : (\phi_1, \phi_1') \uplus_{\phi,\phi} (\phi_2, \phi_2') = ((\phi_1 \uplus_\phi \phi_2), (\phi_1' \uplus_\phi \phi_2'))$$

We distinguish between two types of such mappings: $\phi_I \in \Phi_\perp$, which maps input variables to the security labels of the channels over which their input actions happen, and $\phi_O \in \Phi_\perp$, which maps message names to the labels of the channels over which they are sent.

The formal rules defining the analysis are shown in Figure 6. The two most notable rules in this analysis are (3) and (4), which update the input and output mappings, respectively, with new elements that pair the input variable, y in Rule (3), and output message, y in Rule (4), with the security label of the channel x over which the input and output actions happen.

$$
\begin{array}{ll}
(1)\ \mathcal{A}_S[\![P \textbf{ where } D_1 \ldots D_n]\!]\ \phi_I\phi_O = & \bigsqcup \mathcal{F} \\
& \text{where} \\
& \mathcal{F} = \{(\phi_I, \phi_O), \mathcal{A}_P[\![P]\!]\ \phi_I\phi_O\ \{D_1 \ldots D_n\}\} \\
(2)\ \mathcal{A}_\mathcal{P}[\![\mathbf{0}]\!]\ \phi_I\phi_O\ \Delta & = (\phi_I, \phi_O) \\
(3)\ \mathcal{A}_\mathcal{P}[\![x^\ell(y).P]\!]\ \phi_I\phi_O\ \Delta & = \mathcal{A}_\mathcal{P}[\![P]\!]\ \phi_I[y \mapsto \{\ell\}]\phi_O\ \Delta \\
(4)\ \mathcal{A}_\mathcal{P}[\![\overline{x}^\ell y.P]\!]\ \phi_I\phi_O\ \Delta & = \mathcal{A}_\mathcal{P}[\![\overline{x}^\ell y.P]\!]\ \phi_I\phi_O[y \mapsto \{\ell\}]\ \Delta \\
(5)\ \mathcal{A}_\mathcal{P}[\![\tau.P]\!]\ \phi_I\phi_O\ \Delta & = \mathcal{A}_\mathcal{P}[\![P]\!]\ \phi_I\phi_O\ \Delta \\
(6)\ \mathcal{A}_\mathcal{P}[\![[x = y]P]\!]\ \phi_I\phi_O\ \Delta & = \begin{cases} \mathcal{A}_\mathcal{P}[\![P]\!]\ \phi_I\phi_O\ \Delta, & \text{if } x = y \\ \phi, & \text{otherwise} \end{cases} \\
(7)\ \mathcal{A}_\mathcal{P}[\![(\nu x)P]\!]\ \phi_I\phi_O\ \Delta & = \mathcal{A}_\mathcal{P}[\![P]\!]\ \phi_I\phi_O\ \Delta \\
(8)\ \mathcal{A}_\mathcal{P}[\![P + Q]\!]\ \phi_I\phi_O\ \Delta & = \mathcal{A}_\mathcal{P}[\![P]\!]\ \phi_I\phi_O\ \Delta\ \ \uplus_{\phi,\phi}\ \ \mathcal{A}_\mathcal{P}[\![Q]\!]\ \phi_I\phi_O\ \Delta \\
(9)\ \mathcal{A}_\mathcal{P}[\![p\lfloor x_1 \ldots x_n \rfloor]\!]\ \phi_I\phi_O\ \Delta & = \mathcal{A}_\mathcal{P}[\![unfold(p\lfloor x_1 \ldots x_n \rfloor, \Delta)]\!]\ \phi_I\phi_O\ \Delta
\end{array}
$$

Fig. 6. An I/O Analysis of Normalised Processes

The remaining rules are described as follows. Rule (1) interprets a specification using the operator, $\bigsqcup \mathcal{F}$, which computes the least fixed point of $\mathcal{A}_\mathcal{P}$. Rule (2) returns the same pair of mappings for a null process. Rules (5) and (7) remove input and silent actions and name restrictions as they have no effect on the two mappings. Rule (6) will continue analysing the process if its conditional matching evaluates to an equality, otherwise, it returns the same pair of mappings. Rule (8) distributes the analysis onto the two sides of a choice and combines the results. Finally, Rule (9) unfolds a named process application.

Based on the above analysis, it is possible to define the property that a system is *leaky*, as follows.

Property 1 (A Leaky System). Define a leaky system as one whose specification, S, has an analysis result, $\mathcal{A}_S[\![S]\!]\ \phi_I\phi_O = (\phi'_I, \phi'_O)$, that satisfies the following

condition:

$$\exists x \in bn(S), \ell_1 \in \phi'_I(x), \ell_2 \in \phi'_O(x) : \ \ell_2 \leq_\ell \ell_1$$

A leaky system is therefore characterised as being unable to keep sensitive data received over high-level channels, from being output over low-level channels.

4.1 An Application of the Leakage Analysis

We next show how this analysis can be applied in the context of *Example 1* in the Introduction to model an Internet-of-Things (IoT) publish-subscribe (pub-sub) server, such as the one defined in [2]. A pub-sub server receives data from various IoT devices, e.g. sensors, and outputs those data onto the relevant *topics*. Interested applications, which have already subscribed to those topics, receive the data either through a pull or a push mode of communication. Figure 7 shows an example of an IoT system set-up running an MQTT broker [2] collecting atmospheric humidity data.

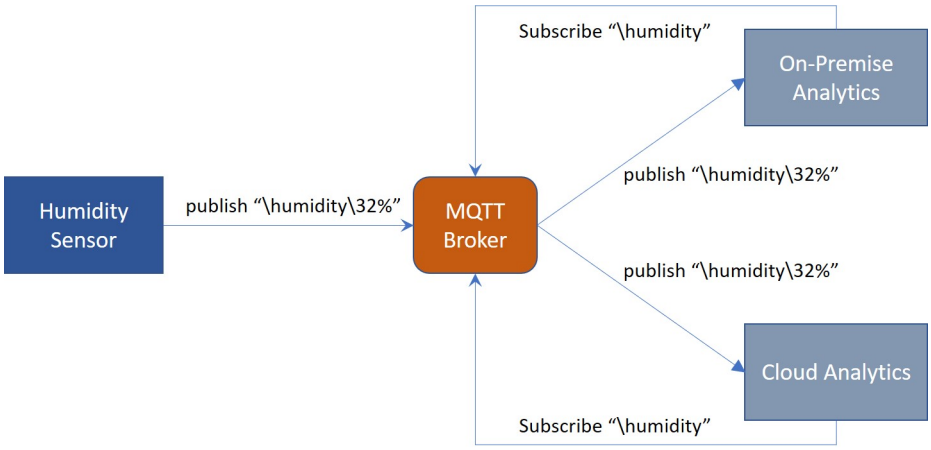

Fig. 7. Example of an IoT (MQTT-based) System Set-up for Collecting Humidity Data

The humidity sensor collects this data and then publishes the data (e.g. humidity level of 32%) to a topic called "humidity". This topic is subscribed to by two software applications: an on-premise (local) analytics application and a cloud-based analytics application that stores data remotely.

In general, a pub-sub server can be modelled as the parallel composition of n-number of pairs of data-processing chained cells:

$$Server \triangleq \prod_{i=1}^{n} (\nu topic)(Data\lfloor sensor, topic \rfloor \mid Data\lfloor topic, application \rfloor)$$

where

$$Data \triangleq (i, o).i(x).\overline{o}x.Data\lfloor i, o \rfloor$$

Each first cell receives data from a *sensor* and outputs those data to an internal *topic*. The second cell then takes the data from that *topic* and outputs it to a subscribed *application*. Every data cell will have similar but distinguished behaviour due to the uniqueness of topics. As was demonstrated in *Example 1*, any of the above individual pairs of cells can be excommunicated into a single cell, as follows:

$$
\begin{array}{l}
\mathit{Data}\lfloor \mathit{sensor}, \mathit{application} \rfloor \\
\textbf{where} \\
\mathit{Data} \triangleq (l, r).l(x).D\lfloor l, r, x \rfloor \\
D \triangleq ((l, r, x).\overline{r}x.\mathit{Data}\lfloor l, r \rfloor + [l = r]D\lfloor l, r, x \rfloor + l(y).\overline{r}x.D\lfloor l, r, y \rfloor)
\end{array}
$$

Returning to the humidity data system of Figure 7, we can model the server in this system as follows:

$$
\begin{array}{rl}
\mathit{Server} \triangleq (\nu \mathit{humidity})(& \mathit{Data}\lfloor \mathit{Humidity\ Sensor}, \mathit{humidity} \rfloor \mid \\
& \mathit{Data}\lfloor \mathit{humidity}, \mathit{On\text{-}Premise\ Analytics} \rfloor \mid \\
& \mathit{Data}\lfloor \mathit{Humidity\ Sensor}, \mathit{humidity} \rfloor \mid \\
& \mathit{Data}\lfloor \mathit{humidity}, \mathit{Cloud\ Analytics} \rfloor)
\end{array}
$$

with the excommunicated version modelled as follows:

$$
\begin{array}{rl}
\mathit{Server} \triangleq (\nu \mathit{humidity})(& \mathit{Data}\lfloor \mathit{Humidity\ Sensor}, \mathit{On\text{-}Premise\ Analytics} \rfloor \mid \\
& \mathit{Data}\lfloor \mathit{Humidity\ Sensor}, \mathit{Cloud\ Analytics} \rfloor)
\end{array}
$$

The latter allowing the humidity sensor to sometimes pass the data to the on-premise analytics application, and sometimes to the cloud-based analytics application, without necessarily revealing the topic being subscribed to.

If we assume that $\ell_2 \leq_\ell \ell_1$ and $\ell_3 =_\ell \ell_1$, and we adopt the following security labelling scheme:

$$
\begin{array}{l}
\mathit{Data}\lfloor \mathit{Humidity\ Sensor}^{\ell_1}, \mathit{On\text{-}Premise\ Analytics}^{\ell_3} \rfloor, \\
\mathit{Data}\lfloor \mathit{Humidity\ Sensor}^{\ell_1}, \mathit{Cloud\ Analytics}^{\ell_2} \rfloor
\end{array}
$$

Then by applying our static analysis, $\mathcal{A}_S[\![\mathit{Server}]\!]\ \phi_I\phi_O$, we obtain the result, $(\phi_I[x \mapsto \{\ell_1\}, y \mapsto \{\ell_1\}], \phi_O[x \mapsto \{\ell_2, \ell_3\}])$, which indicates the presence of information leakage according to Property 1, since $\exists x : \phi_O(x) \leq_\ell \phi_I(x)$.

We note here that our analysis of the excommunicated specification is simpler than if we were analysing the original specification, since we do not have to deal with internal communication, in this case over the *humidity* topic internal channel. In addition, including the *humidity* channel may hide the leakage in the case where there are multiple *Humidity Sensors* and one of these sensors is annotated with a lower label than the *humidity* channel, but still higher than either the *On-Premise Analytics* or *Cloud Analytics* channels. In this case, no leakage would be detected between the *Humidity Sensor* and the *humidity* topic channels, only between *humidity* and the *Cloud Analytics* channels, the latter being too strong a condition as to exclude all *sensor* communication to *application* in our more general definition of the pub-sub system.

5 Conclusion and Further Work

In this paper, we have presented an automatic transformation algorithm which removes all parallel composition, and hence all internal communication, from π-calculus specifications in which all named processes are in a specialised form we call serial form. We have proved that the transformation preserves equivalence, and also that it always terminates. We argue that this transformation facilitates the proving of properties of concurrent systems, and have demonstrated this by showing how it can be used to simplify a leakage analysis.

There are a number of possible directions for further work. Firstly, as was done for the deforestation algorithm [9], an extended version of the excommunication algorithm could be developed that can be applied to all π-calculus specifications. This would involve the identification of terms that could prevent the termination of the algorithm so they are transformed separately. In [9], this required the identification of intermediate data structures within function definitions. Here, it would involve the identification of parallel compositions within named process definitions.

Secondly, a transformation analogous to the supercompilation transformation [18, 16] for functional languages could be developed for the π-calculus. This would involve the use of a homeomorphic embedding relation on processes to determine when generalisation should be performed. Positive information propagation could also be performed using the results of matching operations.

Finally, a transformation algorithm analogous to the distillation algorithm [10] for functional languages could be developed for the π-calculus. Just as the distillation algorithm builds on top of the supercompilation algorithm, this algorithm could build on top of the transformation corresponding to supercompilation for the π-calculus.

References

1. B. Aziz and G.W. Hamilton. A Denotational Semantics for the π-Calculus. In *Proceedings of the 5th Irish conference on Formal Methods*, pages 37–47, 2001.
2. Andrew Banks and Rahul Gupta. MQTT Version 3.1.1 Plus Errata 01. Technical report, OASIS, 2015.
3. Nacéra Bensaou and Irène Guessarian. Transforming Constraint Logic Programs. *Theoretical Computer Science*, 206(1-2):81–125, 1998.
4. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.
5. Nicoletta de Francesco and Antonella Santone. Unfold/Fold Transformations of Concurrent Processes. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 167–181. Springer, 1996.
6. Sandro Etalle, Maurizio Gabbrielli, and Maria Chiara Meo. Transformations of CCP programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):304–395, 2001.
7. M. Gengler and M. Martel. Self-Applicable Partial Evaluation for the π-Calculus. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 36–46, 1997.

8. G. W. Hamilton. Higher Order Deforestation. In *Proceedings of the Eighth International Symposium on Programming, Logics, Implementation and Programs*, pages 213–227, 1996.
9. G. W. Hamilton. Extending Higher Order Deforestation: Transforming Programs to Eliminate Even More Trees. In *Trends in Functional Programming (Volume 3)*, pages 25–36. Intellect Books, 2002.
10. G.W. Hamilton. Distillation: Extracting the Essence of Programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 61–70, 2007.
11. Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial Evaluation Scheme for Concurrent Languages and its Correctness. In *European Conference on Parallel Processing*, pages 625–632. Springer, 1996.
12. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, 1992.
13. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, II. *Information and Computation*, 100(2):41–77, 1992.
14. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
15. Dan Sahlin. Partial evaluation of AKL. In *Proceedings of the First International Conference on Concurrent Constraint Programming*, 1995.
16. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
17. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Second International Conference on Logic Programming*, pages 127–138, 1984.
18. V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):90–121, July 1986.
19. Kazunori Ueda and Koichi Furukawa. *Transformation rules for GHC Programs*. Institute for New Generation Computer Technology Tokyo, 1988.
20. P.L. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.